DOI 10.12694/scpe.v17i2.1162
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 2, pp. 143–153. http://www.scpe.org

PROVENANCE BASED CHECKPOINTING METHOD FOR DYNAMIC HEALTH CARE SMART SYSTEM*

ESZTER KAIL† KRISZTIÁN KARÓCZKAI‡ PÉTER KACSUK§ AND MIKLÓS KOZLOVSZKY¶

Abstract. Smart systems in telemedicine frequently use intelligent sensor devices at large scale. Practitioners can monitor non-stop the vital parameters of hundreds of patients in real-time. The most important pillars of remote patient monitoring services are communication and data processing. Large scale data processing is done mainly using workflows. Some workflows are working in real-time, more complex ones are running for days or even for weeks on parallel and distributed infrastructures such as HPC systems and cloud. In HPC environment high number of failures can arise during health care smart systems workflow enactment, so the use of fault tolerance techniques is unavoidable. The most frequently used fault tolerance technique is checkpointing. The effectiveness of the checkpointing method depends on the checkpointing interval. In this work we give a brief overview of the different checkpointing techniques and propose two new provenance based checkpointing algorithms which uses the information stored in the workflow structure to dynamically change the frequency of checkpointing and can be efficiently used for dynamic health care smart systems.

Key words: scientific workflow, health care, fault tolerance, checkpointing

AMS subject classifications. 68M14

1. Introduction. Smart systems are by definition miniaturized devices that incorporate functions of sensing, actuation and control. Telemedicine frequently uses intelligent sensor devices at large scale. The deployed health care devices (such as ECG, pulse meters, blood glucose meters, etc.) at patient's home are firstly sensing then later transmitting huge amount of data, thus building up communication intensive smart system networks. A single practitioner can easily monitor vital parameters of hundreds of patients 24/7 in real time. Remote patient monitoring has several advantage as: the patient is monitored location independently, not restricted physically to a single place, patient is not connected tightly to the doctor, or to the hospital, on the contrary patients can be monitored during their everyday life, within their normal environment, medication can be adapted to the patient's normal lifestyle and not to an artificial situation when they are hospitalized. The most important pillar of telemedicine services is communication. Well matured technologies, practices, industry standards and reliable infrastructure are available for sensor data transport and storage.

The medical diagnosis requires more than just simple data collection and visualization of sensor data. The sensor data should be pre/post-processed (data filtering, cleaning and analysis of basic sensor data patterns). Sensor data processing is by definition a complex task, and it is mainly a special Big Data challenge both for engineers and for data processing systems. The elementary data size can vary from the very small as a few bytes to some thousands of bytes, however, if we look at the number of already available data sources and their data sending frequency, it turns out, that even simple real-time health care monitoring tasks (e.g.: measure pulse, 6-12 channels ECG -600 Hz-, and SpO2) requires more processing resources than conventional systems can effectively handle.

Data processing workflows used in health care smart systems are data and computational intensive, thus they may require long execution time, which in certain cases can even last for days. During long intervals it is inevitable to adapt to the dynamically changing environment which can be caused by unwanted input data, crash faults or network problems. In one of our earlier works [5] we defined the main requirements of dynamic workflow execution systems as: the ability to react to or to handle unforeseen scenarios raised during the workflow enactment phase, to adapt to new situations, to change the abstract or concrete workflow model

*This work was supported by EU project SCI-BUS

†Óbuda University, John von Neumann Faculty of Informatics, Biotech Lab Bécsi str. 96/b., H-1034, Budapest, Hungary (kail.eszter@nik.uni-obuda.hu).

‡MTA SZTAKI, LPDS, Kende str. 13-17, H-1111, Budapest, Hungary

§MTA SZTAKI, LPDS, Kende str. 13-17, H-1111, Budapest, Hungary and University of Westminster, 115 New Cavendish Street, London W1W 6UW

¶Óbuda University, John von Neumann Faculty of Informatics, Biotech Lab Bécsi str. 96/b., H-1034, Budapest, Hungary and MTA SZTAKI, LPDS, Kende str. 13-17, H-1111, Budapest, Hungary

or to give faster execution and higher level performance according to the actual environmental conditions and intermediary results. We can define three main areas of dynamism which are optimization of the workflow execution according some criteria, user-steering (user or administrator interaction during execution) and fault tolerance behavior [4]. In this work we delve mainly into fault tolerance behavior.

Fault tolerance is the ability of a system to perform its functions even in the presence of a failure [2]. There are two main groups of failures that could arise during enactment. The first group includes the crash faults or fail-stop faults which may come with faulty system components that result in complete data loss. The other group consists of byzantine faults which result the system components to behave unpredictably and maliciously. Byzantine failures can occur, e.g., due to software bugs, (transitional or permanent) hardware malfunction, or malicious attack. In our work we consider only crash faults where the complete state of the actual task and environment must be restored.

Before we go more detailed into fault tolerance we need to define three important terms:

- Checkpointing is a technique to save the complete status of an executing program or job and to restore and restart from this so called checkpoint (which is basically a snapshot of the program's state) later if the original program or job was halted due to system failures.
- Breakpoint: is an intentional stopping or pausing place in a program. It is used extensively by software developers and testers to stop the program whenever a certain point in the program is reached. For each breakpoint, conditions can be added to control in finer detail whether the program should stop.
- Watchpoint: It is a special conditional breakpoint that stops the program when the value of an expression changes. Watchpoints are not set for functions or lines of code, but are set on variables. When those variables are read or written, the watchpoint is triggered and program execution stops.

Fault tolerance policy can be reactive and proactive. While the aim of proactive techniques is to avoid situations caused by failures by predicting them and taking the necessary actions, reactive fault tolerance policies reduce the effect of failures on application execution when the failure effectively occurs. Most of the scientific workflow management systems are working reactively (e.g.: gUSE WMS), and steering workflows according to the available situation. There are several solutions in the literature for fault tolerant behavior and other complementary methods in its connected fields [1]. To achieve fault tolerant behavior the most widely adopted methods are:

- Checking and monitoring, which is a key factor in failure detection.
- Checkpointing and resubmission, where the system state is captured and saved based on predefined parameters (i.e.: time interval, number of instructions) and when the system undergoes some kind of failure the last consistent state is restored and computation is restarted from that point on [2].
- Replication, where critical system components are duplicated using additional hardware, or with scientific workflows critical tasks are replicated and executed on more than one processor. We can differentiate active and passive replication. Passive replication means that only one primary processor is invoked in the execution of a task and in the case of a failure the backup ones take over the task processing. In the active form all the replicas are executed at the same time and in the case of a failure the replica can continue the execution without intervention. The idea behind task replication is that replication size r can tolerate $r-1$ faults while keeping the impact on the execution time minimal. We call r the replication size [12].
  While this technique is useful for time-critical tasks its downsides lays in the large resource consumption, so our attention is focused on mainly checkpointing methods in this work.

1.1. Motivation. Our main aim is to realize a more dynamic workflow management system to decrease overall processing/checkpointing time at the workflow level. In this paper we build up an Adaptive Provenance Based (APB) checkpointing model and propose two new checkpointing algorithms. Provenance in general carries information about the source, origin, and processes that are involved in producing data. The main target of collecting provenance data is support driven thus at the end the system effectively provides reusability and reproducibility for the user/scientist/developer communities. However, provenance can also support fault tolerant behavior by providing statistics about historical executions, such as failure rates or distribution and by storing the intermediary results generated by each tasks of the workflow [3].

To analyze the effects of the length of the checkpointing interval more precisely, we have based our approach on the fact, that the different paths constituting the workflow may have different time requirements

and constraints, moreover longer checkpointing intervals may cause longer rework time in the case of a failure. For example if the workflow consists of only a single task or two or more sequentially ordered tasks (Fig. 1.1) then the aim can be the minimal execution time or to meet a soft or hard deadline. In this case the deadline determines whether we should use optimal checkpointing interval, which expectedly result in minimal execution time, or we have a little spare time where the fault tolerance parameters can be adjusted.

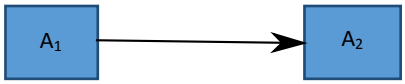

FIG. 1.1. *Workflow with two sequentially ordered tasks*

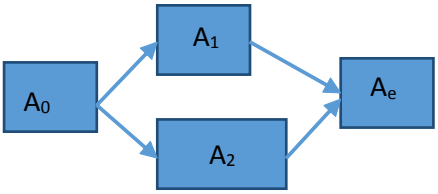

FIG. 1.2. *Workflow with two parallel executable, heterogeneous tasks*

Similarly if we have two or more parallel paths in the workflow model as in Fig. 1.2 we can take advantage of the different time requirements and constraints concerning the individual paths. Arising a failure during task A_1 or task A_2 the effect of the failure can be different, thus different fault tolerant parameters can be used.

Our main contribution is to dynamically adapt the checkpointing frequency based on the workflow structure in order to minimize the overall cost of checkpointing in time, network usage and storage capacity while still keeping to the soft or hard deadline.

1.2. Paper Organization. Our paper is organized as follows. After the introduction we give an overview about the existing theoretical checkpointing methods, and we also introduce some examples about the used solutions in the existing scientific workflow management systems. In Section 3 we introduce and detail our checkpointing method with algorithmic description. In Section 4 we provide use cases focusing on different (more-and-more complex) workflow structures and we also demonstrate the usefulness of our algorithms in an often used health care workflow. Finally, after the conclusions the bibliography closes our work.

2. Related Work.

2.1. Theoretical Model. Concerning dynamic workflow execution fault tolerance is a very important issue and checkpointing is the most widely used methods to achieve fault tolerant behavior. We investigated the different algorithms in order to give a brief overview of them.

According to the level where the checkpointing occurs we differentiate:

- application level checkpointing,
- library level checkpointing,
- system level checkpointing methods.

Application level checkpointing means that the application itself contains the checkpointing code. The main advantage of this solution lies in the fact, that it does not depend on auxiliary components however it requires a significant programming effort to be implemented while library level checkpointing is transparent for the programmer. Library level solution requires a special library linked to the application that can perform the checkpoint and restart procedure. System level solution can be implemented by a dedicated service layer that hides the implementation details from the application developers but still give the opportunity to specify and apply the desired level of fault tolerance [6].

From another perspective we can differentiate coordinated and uncoordinated methods. With coordinated checkpointing (synchronous) the processes will synchronize to take checkpoints in a manner to ensure that the

resulting global state is consistent. This solution is considered to be domino-effect free. With uncoordinated checkpointing (independent) the checkpoints at each process are taken independently without any synchronization among the processes. Because of the absence of synchronization there is no guarantee that a set of local checkpoints result in having a consistent set of checkpoints. It may lead to the initial state due to domino-effect.

The frequency of the checkpointing interval also imposes many opportunities in checkpointing algorithms. Young in [7] has already in 1974 defined his formula for the optimum periodic checkpoint interval which is based on the checkpointing cost and the mean time between failures (MTBF) with the assumption that failure intervals follow an exponential distribution.

Di et al in [8] has also derived a formula to compute the optimal number of checkpoints for jobs executed in the cloud. His formula is generic in a sense that it does not use any assumption on the failure probability distribution.

The drawback of these solutions lies in the fact that the checkpointing cost can change during the execution if the memory footprint of the job changes, or depending on network reachability issues or when the failure distribution changes. Thus static intervals may not lead to the optimal solution. By dynamically assigning checkpoint frequency we can eliminate unnecessary checkpoints or where the danger of a failure is considered to be severe it can introduce extra state savings.

Meroufel and Belalem [9] proposed an adaptive time-based coordinated checkpointing technique without clock synchronization on cloud infrastructure. Between the different VMs jobs can communicate with each other through a message passing interface. One VM is selected as initiator and based on timing it estimates the possible time interval where orphan and transit messages can be created. There are several solutions to deal with orphan and transit messages, but most of them solve the problem by blocking the communication between the jobs during this time interval. However blocking the communication increases the response time and thus the total execution time of the workflow which can lead to SLA violation. In Meroufel's work they avoid blocking the communication by piggybacking the messages with some extra data so during the estimated time intervals it can be decided when to take checkpoint or logging the messages can resolve the transit messages problem.

The initiator selection is also investigated in Meroufel and Belalems another work [10] and they found that the impact of initiator choice is significant in term of performance. They also propose a simple and efficient strategy to select the best initiator.

Di et al also propose a new adaptive algorithm to optimize the impact of checkpointing regarding the checkpointing or restarting costs in [8].

Theresa et al in their work [11] propose two dynamic checkpoint strategies: Last Failure time based Checkpoint Adaptation (LFCA) and Mean Failure time based Checkpoint Adaptation (MFCA) which takes into account the stability of the system and the probability of failure concerning the individual resources.

This paper introduces a novel checkpointing method that is based on workflow model structure and failure statistics gathered about resources from historical executions. It extends related work on workflow structure analyses which focuses mainly on workflow similarity issues concerning the efficient storing and sharing of reproducible workflows [18], on scheduling mechanisms, and also on workflow execution time estimation problems [17]. Our work also promotes researches in fault tolerance by including the information obtained from the workflow structure into the actual state analysis and thus into the checkpointing interval determination.

2.2. Model implementations. Most of the existing workflow management systems are using some sort of mechanism to introduce fault tolerance in the system. As an example we are listing here two approaches:

2.2.1. gUSE. gUSE [16] (grid and cloud user support environment) is an open source science gateway (SG) framework, developed by Laboratory of Parallel and Distributed Systems (LPDS) at the MTA SZTAKI. It provides a generic purpose, workflow-oriented graphical user interface to create and run (DAG like) workflows on various Distributed Computing Infrastructures (DCIs) including clusters, grids, desktop grids and clouds.

- gUSE breakpoint support
  In gUSE breakpoints can be assigned to a job in two places: before submission of a job and after termination of a job. In the former case the user can check and modify the next running job instance of the workflow instance directly before starting it, while in the latter case the user has additional information about the job instance. A waiting time can also be set concerning the actual breakpoint.

- gUSE checkpointing support
  The suspend operation executes a job instance level check pointing, saving only those job items, which terminated properly (being in state finished) and sets back the system to be ready to continue its working, using the outputs of the properly terminated job instances.

2.2.2. Taverna. Taverna [15] is an open source and domain-independent Workflow Management System, developed by the myGrid team. It provides a desktop authoring environment (Taverna Workbench) and enactment engine for scientific workflows.

- Taverna breakpoint support
  Taverna has breakpoint support, including the editing of intermediate data. Breakpoints can be placed during the construction of the model at which execution will automatically pause or by manually pausing the entire workflow. However in Taverna the e-scientist cannot find a way to dynamically choose other services to be executed on the next workflow steps depending on the results.

2.2.3. Pegasus. The Pegasus system [14] was developed at the University of Southern California, with the aim to help workflow-based applications to be executed in a number of different environments including desktops, campus clusters, grids, and clouds.

- Pegasus checkpoint support
  Pegasus uses a detailed provenance database to keep track of what has been done including the locations of data used and produced, and which software was used with which parameters. When errors occur, Pegasus tries to recover when possible by retrying tasks, and when all else fails, generates a rescue workflow containing a description of only the work that remains to be done. A rescue DAG only skips jobs that have completely finished. It does not continue a partially running job unless the executable supports checkpointing. With this rescue DAG Pegasus implements a workflow level checkpointing similar to the suspend feature of the gUSE system.

3. Proposed Model - APB Chekpointing. To achieve a more dynamic workflow management system which is capable to effectively resolve faults during program execution we apply a provenance based checkpointing method (Adaptive Provenance Based - APB Checkpointing) at workflow management system level. We are focusing on DAG (directed acyclic graph, which is a directed graph with no directed cycles) based workflow enactment such as used in gUSE. We have identified the following environmental conditions where the proposed APB model can work effectively.

3.1. Environmental Conditions.

- The system resources are monitored and failures can be detected as soon as possible, therefore the fault detection time (t_f) does not add high latency to the overall makespan of the workflow execution ($t_f = 0$ considered during our research).
- Task A_j cannot be started before it has received the output from all its predecessors and the results of a Task A_i can only be sent to its successor tasks after the task has been finished. Concerning a simple workflow as in Fig. 1.1 task A_e can only be started after the successful termination of both tasks A_1 and A_2.
- There is an ideal case so that tasks can be executed as soon as all the results from the predecessor tasks are ready and available. The system resources are inexhaustible in number, so the system can allocate the required number of resources to execute all the tasks parallel that are independent from each other.
- The system supports the collection of provenance data, therefore the intermediary results generated by the individual tasks are saved and in case of failure they can be easily retrieved. Thus there is no need to take checkpoints at the end of the tasks, and there is no need to take global checkpoints, since in the case of failure only the effected task should be rolled back.
- The system also support provenance data about failure statistics, so the probability of failures for a certain period of time is available for each resource component taking into account the aging factor as well.

3.2. General notation. Workflows in general and also scientific workflows can be represented as directed acyclic graphs (DAGs) $W = (N, E)$, where the nodes (N) represent the computational tasks or jobs and the

directed edges (E) represent the dependency between them. The dependency can be data dependency, and control dependency. Scientific workflows are mainly data driven, so we are focusing on data flow oriented workflows in our paper. In this case the output of a Task A_i gives the input of a Task A_j if there exists an $A_iA_j \in E$ directed edge in the workflow. The list of symbols used in this paper can be seen in Table 3.1.

TABLE 3.1
Notation of used symbols

T_c	The checkpointing interval
T_{opt}	The optimal checkpointing interval
X_i	Optimal number of checkpoints during the execution of a task A_i
C	Checkpointing cost (considered constant)
$T(A_i)$	Execution time of task with T_{opt} and $E(Y)$
T_f	Mean time between failures (MTBF)
$E(Y)$	Expected number of failures during the execution of a task
T_l	Loading time, to restore the last saved checkpoint state
t_f	Fault detection time, the time to detect the failure
A_0	First or entry task of the workflow
A_e	Last or exit task of the workflow

For our APB model we propose two new checkpointing algorithms that with the help of monitoring the resources and executions dynamically adjust the checkpointing interval based on the task's dependency factor and the already occurred failures.

In the proposed algorithms there is no need to take global checkpoints of the workflow, and therefore there is no need of synchronization of any kind (based on time or based on communication channels between the processors). The parallel threads of the workflow may run on different type of computing infrastructures (for example on virtual machines of different cloud providers) therefore it would be a complex challenge to solve the synchronization between them.

3.3. Algorithms. The primary goal of our algorithms is to minimize the checkpointing overhead (time, resource) while still keep to the soft-deadline of the workflow and the performance level at a satisfactory level.

Young [6] and Di [7] have already proved that the optimal checkpointing interval, or the optimal number of checkpoints concerning the execution of a single task can be computed by Equation 3.1 and 3.2.

In their investigations Di et al. have declared that with equidistant checkpointing model, constant checkpointing cost (referred to as time overhead and denoted by C) and task restarting cost the total wallclock time of a task can be written as the sum of the calculation time, the checkpointing costs and the recovery time after failures. Their calculation was based on the assumption that the rework time after a failure of a task is about half of the checkpointing interval. By minimizing the expected time of execution they get their optimal value.

In both equations the fault detection time is considered $t_f = 0$. Equation 3.1 is a more general form, because it does not depend on any probability distribution, unlike the Young (3.2) formula which needs to assume that failure intervals follow an exponential distribution.

In both equations C is the checkpointing cost, T_{opt} is the checkpointing interval, T_f is the mean time between failures, $T(A_i)$ is the expected time of execution, $E(Y)$ is the expected value of failures during the execution of a task.

$$X = \sqrt{\left(T(A_i) \cdot \frac{E(Y)}{2C}\right)}, \tag{3.1}$$

$$T_{opt} = \sqrt{2CT_f}. \tag{3.2}$$

Our first algorithm is a static solution, while the other one is an adaptive one. The main difference between them lies in the fact, that the static algorithms does not change the checkpointing interval during the execution

of the workflow it only adjust the length of the intervals before workflow submission. The second, adaptive solution may adjust the frequency of the checkpointing before each task execution. While the first algorithm gives adaptivity at workflow level, the other one proposes a real task level adaptive solution.

In our proposed algorithms we use (3.1) as a starting point to compute the checkpointing intervals. The main idea is that there is a dependency factor between the tasks. Namely if a failure occurs during the execution of a task A_i then it not only has a local effect on the task itself, but has a global effect also on the whole workflow concerning the execution time. Since if a failure occurs during the execution of task A_i then it has to be re-executed from the last checkpoint. It means the execution of the task ends later, so it may cause all of the successor tasks of task A_i to wait for the results. This can result the whole workflow execution to last longer. Similarly if we decrease the frequency of checkpointing, in other words we increase the length of the checkpointing interval T_c, then it has also a local effect, but the scope of its effect can extend for more tasks or even for all tasks as well.

3.3.1. Static Workflow Level (SWL) Algorithm. We define local cost (3.3) of increasing the checkpointing interval of task A_i, which is the execution time overhead of a task if the number of checkpoints is decreased by 1:

$$C_{local} = \frac{ET(A_i) + (\frac{T_c - T_{opt}}{2}) \cdot E(Y) - C}{ET(A_i)}. \tag{3.3}$$

We define the global cost (3.4) of increasing the checkpointing interval of a task A_i, which is the execution time overhead of the whole workflow, if the number of checkpoints of task A_i is decreased by 1:

$$C_{global} = \frac{(\frac{T_c - T_{opt}}{2}) \cdot E(Y) - C + rank(A_i) + brank(A_i)}{rank(A_0)}, \tag{3.4}$$

where the $rank()$ (3.5) function is a classic formula that are used in tasks scheduling [12] [13]. Basically the $rank()$ function is the critical path from task A_i to the last task, and can be computed recursively backward from the last task. For simplicity we have introduced the $brank()$ (3.6) function, which is the backward $rank()$ value from task A_i backward to the entry task A_0. It is the longest distance from the entry task to task A_i excluding the computation cost of the task itself. It can also be calculated recursively downward from task A_0.

$$rank(A_i) = T(A_i) + max_{A_j \in succ(A_i)} \cdot rank(A_j), \tag{3.5}$$

$$brank(A_i) = max_{A_j \in pred(A_i)} \cdot (brank(A_j) + T(A_j)). \tag{3.6}$$

Before submitting the workflow at first the optimal checkpointing interval should be calculated for each task based on the failure statistics of the resource (expected value of the failures that can arise during execution) and the estimated (or retrieved from provenance database) execution time of the task. Than the adjusted checkpointing intervals for all tasks can be calculated. Concerning those tasks that are part of the critical path or one of the critical paths of the workflow the checkpointing interval should remain the optimal value. However along all other paths between A_0 and A_e in the abstract model of the workflow the checkpointing interval should be adjusted. If inequality 3.7 stands for a task A_i it means, that for that task the checkpointing interval can be increased without effecting the total wallclock time of the workflow. The cumulative costs along a path should be taken into account.

The pseudocode of the WLS static algorithm can be seen in Table 3.2 and WLS works as follows: The algorithms analyzes the tasks in a so called topological ordering. In a topological ordering task A_i precedes A_j if during execution task A_i should also precede task A_j because task A_j or any predecessors of task A_j uses its results as an input parameter. WLS is a fair algorithm, it tries to share the spare time equivalently. Given a path s_j between A_0 and A_e the algorithm makes attempts to adjust the checkpointing interval of the tasks belonging to this path in execution ordering

TABLE 3.2
Pseudo code of the WLS algorithm

```
FOR i : 1 TO N DO
  T_adj(A_i) = T(A_i)
  brank_adj(A_i) = max_{A_j ∈ pred(A_i)} · (brank_adj(A_j) + T_adj(A_j))
While ∃i:  X_i > 1 AND brank_adj(A_i) + rank(A_i) < rank(A_0) DO
  FOR i : 1 TO N DO
    IF brank_adj(A_i) + rank(A_i) < rank(A_0) THEN DO
      T_adj(A_i) = C_local
      X_i = X_i − 1
    ELSE T_adj(A_i) = T(A_i)
```

$$brank(A_i) + rank(A_i) + \frac{T_c - T_{opt}}{2} \cdot E(Y) - C < rank(A_0). \tag{3.7}$$

The first three lines of the algorithm in (Table 3.1) initializes the modified execution time $T_{adj}(A_i)$ and cumulative execution time $brank_{adj}(A_i)$ values concerning the individual paths based on the optimal checkpointing interval. Next, in topological ordering the checkpointing intervals are adjusted. When the checkpointing intervals for all tasks have been adjusted once, the algorithms repeats it until the workflow structure enables it.

This algorithm is based on the assumption that when faults occur during a checkpointing interval (between two consecutive checkpoints) the expected average time loss is half of the checkpointing interval and also based on the a-priori calculated (or from provenance retrieved) expected values of failures and execution times of the individual tasks. This algorithm is executed once before submitting the workflow and after that the checkpointing intervals are not modified. It gives a workflow level, static solution for optimizing checkpointing costs.

3.3.2. Adaptive task level (ATL) algorithm. If the expected number of failures are already met during the execution of the actual task or this was the case during one of the predecessors of the actual task, then it may be possible, that default checkpointing intervals should not have been increased, because the cumulative overhead of the occurred faults can negatively affect the whole workflow execution time. Or the shoe is on the other foot, the predecessors have been executed without or less failures and thus a lot of free time remained to (expectedly) meet the soft or hard deadline and thus to minimize the overhead caused by frequent checkpointing. To take earlier faults and real execution conditions (changing failure distributions or execution times) into account we need information about the realistic execution time of the tasks. With provenance support the real execution time and actual failure statistics can be obtained and the algorithm can adjust the checkpointing interval dynamically. This adaptive algorithm is based on the assumption that failures are detected as soon as possible, and system reacts immediately.

Before executing the individual tasks the checkpointing intervals are adapted to the new situation.

The ATL adaptive algorithm works as follows. Before submitting a task A_i the estimated $brank(A_i)$ values are updated with real execution time of the predecessor tasks and based on the actual values of the critical path or paths the checkpointing intervals for the remaining tasks of the workflow is recalculated.

4. Use Cases. As it was already shown in the previous section the effect of the length of the checkpointing interval concerning the individual tasks can be different on the whole workflow based on the assumption that when a failure occurs during the execution of a task the average rework time is half of the checkpointing interval.

4.1. Basic workflow structures. Concerning a single task workflow or a workflow, constituting of sequentially ordered tasks, when we increase the length of the checkpointing interval it surely has a global effect on the makespan of the workflow. However, if soft or hard deadline T_D is given, and $T(A_i) < T_D$ then we have the opportunity to reduce the number of checkpoints, so that execution time still remains less or equal then the predefined deadline. Thus in this simple case both the static (SWL) and adaptive (ATL) algorithms can be used.

Concerning a workflow with two parallel executable, heterogeneous tasks (1.2) we can differentiate two cases:

- If $T(A_2) >> T(A_1)$ than A_2 should be executed with optimal checkpointing and A_1 can be executed without any checkpoints. If A_1 fails for the first time it has still enough time to be executed once more with minimal checkpoints, until the remaining time is only enough for an optimal execution.
- If $T(A_2) > T(A_1)$ than A_2 should be executed with optimal checkpointing and A_1 should be executed with less frequent checkpoints (static (SWL) algorithm) or with adaptive checkpointing (ATL) algorithm).

Concerning a workflow with two parallel executable, homogeneous tasks as a starting point we do not have spare time because the other task may generate its result on time. The only way we may minimize the checkpointing overhead is to use the adaptive (ATL) algorithm. The two tasks are started with optimal checkpointing and when a failure occurs during the execution of one of the two tasks then the other task can use the adaptive algorithm with the increased makespan of the first task as a deadline.

Sequentially and parallel ordered tasks can form arbitrary constructed workflow. As our algorithms calculates the *brank*() value along the paths of the workflow starting from the entry task A_0 to the exit task A_e, it can be used for arbitrary constructed DAGs.

4.2. Dynamic Health Care Workflow. The workflow depicted in Fig. 4.1 is often used in health care smart systems.

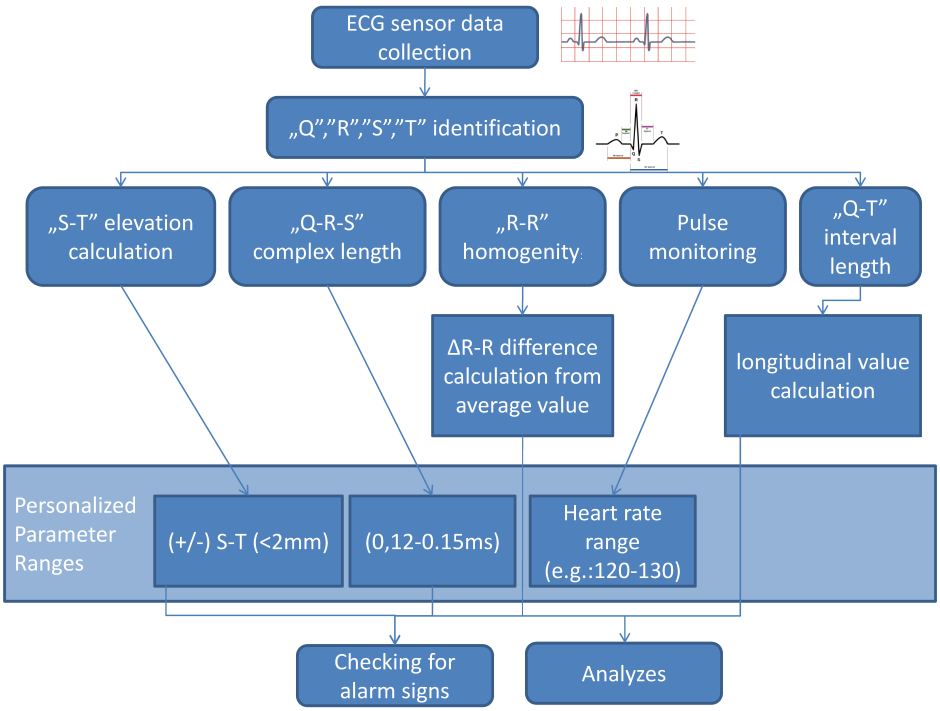

Fig. 4.1. *Cardiac Workflow*

4.2.1. Workflow description. Workflow inputs: ECG sensor data and general+personalized parameter ranges: (normal/green zone/, small parameter value deviation/yellow zone/, large parameter value deviation /red zone/).

Workflow outputs: inform external system (alarm signal/no alarm signal), analyzed results.

The cardiac workflow receives the acquired ECG sensor data and automatically identify special landmark points on the ECG curves. As a next step it calculates special parameters like the QRS complex length, or

QT interval. These small calculations have to be done mostly at each ECG signal cycle. After the calculations the parameter values are evaluated against the pre-defined parameter ranges. These (general and personalized) parameter ranges specify what is a ,,normal" parameter value (defined as green zone), what is slightly (defined as yellow zone) or significantly abnormal value (defined as red zone).

The result of the parameter value evaluation is the basic input of an external decision support application which can realize actions (such as alarming) according to the received information.

4.2.2. Usability of our checkpointing algorithm. The workflow depicted in Fig. 4.1 can be used in numerous scenarios. For example to monitor heart activity for a smaller period of time (for monitoring patients after stroke during physical training or load, or monitoring professional football players during a match), but also for monitoring longer periods of time. One of its long-term usage is the Holter ECG. Holter ECG monitor records electrical signals from the heart via a series of electrodes attached to the chest during 24 h period of time or even during several months as well. The number and position of electrodes varies by model, but most Holter monitors employ between three and eight. These electrodes are connected to a small piece of equipment that is attached to the patient's belt or hung around the neck, and is responsible for keeping a log of the heart's electrical activity throughout the recording period. There are also available intelligent systems that already make some analyzes on the recorded data, but the widespread used model are only capable to log data and after the measurement period to send these data to a computing resource where these data are analyzed. In general a 24 h interval is logged on Holter appliances. At the time the cardiologist retrieves the ECG sensor data collection from the Holter, he or she needs the analyzed results very fast, so the workflow should be enacted as fast as it can, and in general it is calculated not for a single patient, but for a group of patients or for even a big population used for cardiological researches.

The workflow model consists of several distinct paths that can be enacted separately and parallel. These distinct paths need different calculation times so our algorithm can be efficiently used here without increasing the total makespan of the workflow execution.

5. Conclusion. Smart systems in telemedicine frequently use intelligent sensor devices at large scale. Their data processing tasks are often realized in workflows. In HPC environment high number of failures can arise during workflow enactment, so the use of fault tolerance techniques is unavoidable. In this paper we investigated the different checkpointing techniques, which are the most widely used proactive fault tolerant methods. We gave a brief overview of the different checkpointing perspectives with special attention on those solutions where the checkpointing intervals are periodic or it changes adaptively during execution. We proposed a workflow level (SWL) and a task level (ATL) provenance based dynamic algorithm where the checkpointing frequency is adjusted in order to eliminate blind checkpoints while still maintaining soft deadlines.

We have compared the two algorithms and the ATL algorithm outperforms the SWL one in adaptivity support, since the static algorithm cannot adjust the checkpointing intervals to the actual environmental conditions (failures, execution times). SWL cannot guarantee soft or hard deadline if the number of failures exceeds the expected value. However, the ATL algorithm is more compute intensive (it must be executed before each task submission) and in most of the cases all the required provenance data are not surely available. Moreover, retrieving data from a provenance database may have high overheads as well.

In our future work we will work out a more sophisticated algorithm which based on available statistics of failures can also adaptively determine the length of the forthcoming checkpointing interval that predictively can avoid faults and still keeps the global cost of the failure limited by the predefined soft or hard deadline.

Acknowledgments. This work was supported by EU project SCI-BUS (SCIentific gateway Based User Support). The SCI-BUS project aims to create customized workflow based gateways connected to Distributed Computing Infrastructures (DCIs) fostering HPC and HTC services without the need to deal with the underlying infrastructures' details.

REFERENCES

[1] A. Bala, and I. Chana, *Fault tolerance-challenges, techniques and implementation in cloud computing*, IJCSI International Journal of Computer Science Issues vol. 9, 2012.
[2] R. Garg and A. K. Singh, *Fault Tolerance in grid computing: State of the art*, International Journal of Computer Science & Engineering Survey vol. 1, pp. 88-97, 2011.
[3] E. Kail, A. Bánáti, P. Kacsuk, and M. Kozlovszky, *Provenance based adaptive and dynamic workflows.*, 15th IEEE International Symposium on Computational Intelligence and Informatics, pp. 215-219, IEEE Press, Budapest, 2014,
[4] E. Kail, P., Kacsuk, and M. Kozlovszky, *A Novel Approach to User-steering in Scientific Workflows*, Proceedings of CGW14, 2014
[5] E. Kail, A. Bánáti, K. Karóczkai, P. Kacsuk, and M. Kozlovszky, *Dynamic workflow support in gUSE*, MIPRO, 2014 Proceedings of the 37th International Convention.
[6] R. Jhawar, V. Piuri, and M. Santambrogio, LU-*Fault Tolerance Management in Cloud Computing: A System-Level Perspective*, IEEE Systems Journal 7, vol 2, 2013.
[7] J.W. Young, *A first order approximation to the optimum checkpoint interval*, Communications ACM, 1974.
[8] S. Di, Y. Robert, F. Vivien, D. Kondo, Cho-Li Wang, and F. Cappello, *Optimization of Cloud Task Processing with Checkpoint-Restart Mechanism*, 112. ACM Press, 2013.
[9] B. Meroufel, and G. Belalem, *Adaptive time-based coordinated checkpointing for cloud computing workflows*, Scalable Computing: Practice and Experience, Vol 15, No 2, 2014.
[10] B. Meroufel, and B. Ghalem, *Policy Driven Initiator in Coordination Checkpointing Strategies*, http://www.wseas.us/e-library/conferences/2014/Istanbul/TELEDU/TELEDU-20.pdf.
[11] A. Lidya, S. Therasa, G. Sumathi, and S. A. Dalya, *Dynamic Adaptation of Checkpoints and Rescheduling in Grid Computing*, International Journal of Computer Applications vol. 3, 2010.
[12] L. Zhao, Y. Ren, Y. Xiang, and K. Sakurai, *Fault-tolerant scheduling with dynamic number of replicas in heterogeneous systems*, 12th IEEE International Conference on High Performance Computing and Communications (HPCC), 2010 pp.434,441, 1-3 Sept. 2010.
[13] H. Topcuoglu, S.Hariri, and M. Wu, *Performance-effective and low-complexity task scheduling for heterogeneous computing*, in IEEE Transactions on Parallel and Distributed Systems, vol.13, no.3, pp.260,274, Mar 2002
[14] https://pegasus.isi.edu [acc. 01.02.2016].
[15] http://www.taverna.org.uk [acc. 01.02.2016].
[16] http://guse.hu [acc. 01.02.2016].
[17] I. Pietri, G. Juve, E. Deelman, R. Sakellariou, *Performance Model to Estimate Execution Time of Scientific Workflows on the Cloud* Proceedings of WORKS14
[18] J. Starlinger, S. Cohen-Boulakia, S. Khanna, S.B. Davidson et al., *Layer Decomposition: An Effective Structure-based Approach for Scientific Workflow Similarity*, 2014 IEEE 10th International Conference on e-Science, vol.1, pp.169,176, 20-24 Oct. 2014

Edited by: Karolj Skala
Received: December 21, 2015
Accepted: March 31, 2016